Proof methods for structured corecursive programs

Jeremy Gibbons[1] and Graham Hutton[2]

[1] Oxford Brookes University; email jgibbons@brookes.ac.uk
[2] University of Nottingham; email gmh@cs.nott.ac.uk

Abstract. Corecursive programs produce values of greatest fixpoint types, in contrast to recursive programs, which consume values of least fixpoint types. There are a number of widely used methods for proving properties of corecursive programs, including fixpoint induction, the *take lemma*, and coinduction. However, these methods are all rather low-level, in the sense that they do not exploit the common structure that is often present in corecursive definitions. We argue for a more structured approach to proving properties of corecursive programs. In particular, we show that by writing corecursive programs using an operator called unfold that encapsulates a common pattern of corecursive definition, we can then use high-level algebraic properties of this operator to conduct proofs in a purely calculational style that avoids the use of either induction or coinduction.

1 Introduction

Recursion has long been recognised as a central concept in computing science, with applications ranging from the theoretical foundations of computation [28] to practical programming techniques [5]. In recent years, it has become increasingly clear that the dual, but less well-known, concept of corecursion is just as central to computing [1, 2, 17, 18]. In this article, we explore methods for proving properties of corecursive programs.

As yet, there appears to be no standard definition for the notion of a corecursive program. In this article, we follow [25] and use the term *corecursive program* for a function whose range is a type defined as a greatest fixpoint, such as the type of infinite lists or streams. Dually, we use the term *recursive program* for a function whose domain is a type defined as a least fixpoint, such as finite lists. Of course, these definitions are overly general (for example, they do not require that the functions make use of self reference in any form), but they will suffice for our purposes here.

All the programs in the article are in the lazy functional language Haskell [20]. This language has the convenient property that recursive types are simultaneously both least and greatest fixpoints [11]. Hence, in this article, the terms recursive and corecursive programs simply refer to Haskell functions whose domain and range are recursive types, respectively.

The most widely used method for proving properties of recursive programs is *structural induction*. Unfortunately, this method is not applicable to corecursive programs, because in general such programs do not have an argument over which induction can be performed. Historically, the basic proof method for corecursive programs has been Scott and de Bakker's *fixpoint induction* [8], which arises from the standard denotational semantics of functional programs. Applying fixpoint induction is rather tedious, but for the special case of corecursive programs that produce lists, one can use Bird and Wadler's *take lemma* [4], or the simpler *approximation lemma* [3]. More recently, Gordon [14] and Turner [31] have argued for the use of *coinduction* as the basic proof method for corecursive programs, which arises from work on the operational semantics of functional programs.

In this article we review the above proof methods for corecursive programs, and observe that they are all rather low-level in the sense that they do not exploit the common structure that is often present in corecursive definitions. We argue for a more structured approach to proving properties of corecursive programs. In particular, we show that by writing corecursive programs using an operator called *unfold* that encapsulates a common pattern of corecursive definition, we can then use the high-level *universality* and *fusion* properties of this operator to conduct proofs in a purely calculational style that avoids the use of either induction or coinduction. This approach is dual to the use of *fold* and its associated properties in recursive programming, which has recently been surveyed in [15].

The article is aimed at a reader who is familiar with the basics of recursive programming and structural induction, say to the level of [4, 3]. No prior knowledge of corecursive programming and proof is assumed. For simplicity, we restrict our attention to corecursive programs that produce lists. However, an extended version of the article will show how our approach naturally generalises to a variety of other datatypes.

2 Corecursive programs

In this section we present a number of simple examples of corecursive programs and their properties. To begin with, consider the following Haskell definitions for corecursive functions that produce infinite lists:

$$
\begin{array}{ll}
\mathit{repeat} & :: \alpha \to [\alpha] \\
\mathit{repeat}\ x & = x : \mathit{repeat}\ x \\
\mathit{from} & :: \mathit{Int} \to [\mathit{Int}] \\
\mathit{from}\ n & = n : \mathit{from}\ (n+1) \\
\mathit{iterate} & :: (\alpha \to \alpha) \to \alpha \to [\alpha] \\
\mathit{iterate}\ f\ x & = x : \mathit{iterate}\ f\ (f\ x)
\end{array}
$$

Unwinding these definitions a few steps, we see that:

$$
\begin{array}{ll}
\mathit{repeat}\ x & = x : x : x : x : \ldots \\
\mathit{from}\ n & = n : (n+1) : (n+2) : (n+3) : \ldots \\
\mathit{iterate}\ f\ x & = x : f\ x : f\ (f\ x) : f\ (f\ (f\ x)) : \ldots
\end{array}
$$

From these examples, it is clear that *repeat* and *from* are special cases of *iterate* f, obtained by taking f to be the identity function *id* and the successor function $(+1)$, respectively. That is, we expect to be able to prove the following equations between corecursive programs:

$$\begin{aligned} &\mathit{repeat} = \mathit{iterate}\ \mathit{id} \\ &\mathit{from} \quad = \mathit{iterate}\ (+1) \end{aligned}$$

The *iterate* function encapsulates a general pattern for producing an infinite list, in which the first value is provided as a seed and each subsequent value is generated by applying a function to the previous value. Another such general function is the well-known *map* function, which produces a new list by applying a function to each of the values in a given list:

$$\begin{aligned} &\mathit{map} \quad :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\ &\mathit{map}\ f\ \mathit{xs} = \mathbf{if}\ \mathit{null}\ \mathit{xs}\ \mathbf{then}\ [\,]\ \mathbf{else}\ f\ (\mathit{head}\ \mathit{xs})\ :\ \mathit{map}\ f\ (\mathit{tail}\ \mathit{xs}) \end{aligned}$$

For example:

$$\mathit{map}\ f\ (x_1 : x_2 : x_3 : x_4 : \ldots) = f\ x_1 : f\ x_2 : f\ x_3 : f\ x_4 : \ldots$$

Note that the function *map* f is recursive as well as being corecursive, because it both consumes and produces a list. The definition of *map* above using the list primitives *null*, *head* and *tail* is more verbose than the standard recursive definition in Haskell using pattern matching. However, we choose to define *map* in this manner because it reveals a structure that we will find to be common to many corecursive programs.

In general, the same infinite list may be produced by many different corecursive programs. For example, by again unwinding the appropriate definitions a few steps, we see that we can produce the infinite list

$$f\ x\ :\ f\ (f\ x)\ :\ f\ (f\ (f\ x))\ :\ f\ (f\ (f\ x))\ :\ \ldots$$

by at least three different programs: *tail* (*iterate* $f\ x$), *map* f (*iterate* $f\ x$), and *iterate* f $(f\ x)$. That is, we expect to be able to prove the following equations between corecursive programs:

$$\mathit{tail} \cdot \mathit{iterate}\ f \quad = \quad \mathit{map}\ f \cdot \mathit{iterate}\ f \quad = \quad \mathit{iterate}\ f \cdot f$$

We conclude this section with three further examples of corecursive programs: *copy* x which produces a list of x's of a given length, *digits* which extracts the decimal digits (in reverse order) from a number, and *sort* which implements selection sort using an auxiliary function *delmin* that removes the minimum number from a list, and is straightforward to define:

$$\begin{aligned} &\mathit{copy} \quad :: \alpha \to \mathit{Int} \to [\alpha] \\ &\mathit{copy}\ x\ n = \mathbf{if}\ n == 0\ \mathbf{then}\ [\,]\ \mathbf{else}\ x : \mathit{copy}\ x\ (n-1) \\ &\mathit{digits} \quad :: \mathit{Int} \to [\mathit{Int}] \\ &\mathit{digits}\ n\ = \mathbf{if}\ n == 0\ \mathbf{then}\ [\,]\ \mathbf{else}\ (n\ \mathbf{mod}\ 10)\ :\ \mathit{digits}\ (n\ \mathbf{div}\ 10) \\ &\mathit{sort} \quad :: [\mathit{Int}] \to [\mathit{Int}] \\ &\mathit{sort}\ \mathit{ns}\ = \mathbf{if}\ \mathit{null}\ \mathit{ns}\ \mathbf{then}\ [\,]\ \mathbf{else}\ \mathit{minimum}\ \mathit{ns}\ :\ \mathit{sort}\ (\mathit{delmin}\ \mathit{ns}) \end{aligned}$$

The structure of these definitions is clearly similar to that for *map*. In fact, we will see later on that all the functions defined above are instances of a common pattern of corecursive definition, and that we can exploit this fact when proving properties of these functions. Prior to this, however, we review and compare the standard proof techniques for corecursive programs.

3 Fixpoint induction

Recall that in the normal denotational approach to the semantics of functional languages such as Haskell [29], types are partially ordered sets with a least element $\bot$ and limits of all non-empty chains (that is, types are *cpos*), and programs are functions between cpos that preserve these limits (that is, programs are *continuous functions*). In this setting, the meaning of a definition $x = f\ x$ for a value x in terms of itself and some auxiliary function f is given by the limit of the following chain:

$$\bot \sqsubseteq f\ \bot \sqsubseteq f\ (f\ \bot) \sqsubseteq f\ (f\ (f\ \bot)) \sqsubseteq \ldots$$

The well-known fixpoint theorem [7] shows that the limit of this chain, denoted by $\mathit{fix}\ f$, is the least solution to the equation $x = f\ x$ (that is, $\mathit{fix}\ f$ is the *least fixpoint* of f), and is hence an appropriate choice for the meaning of x. As an example of this approach to semantics, recall the defining equation for the corecursive program *repeat* given earlier:

$$\begin{array}{l} \mathit{repeat} \quad :: \alpha \to [\alpha] \\ \mathit{repeat}\ x = x : \mathit{repeat}\ x \end{array}$$

This definition can be rewritten as $\mathit{repeat} = f\ \mathit{repeat}$, where the auxiliary function f is defined without reference to itself as follows:

$$\begin{array}{l} f \qquad :: (\alpha \to [\alpha]) \to \alpha \to [\alpha] \\ f\ g\ x = x : g\ x \end{array}$$

Hence, the meaning of *repeat* is given by $\mathit{repeat} = \mathit{fix}\ f$, from which a simple calculation shows that $\mathit{repeat}\ x$ is the limit of the following chain of partial lists containing increasing numbers of x's, as expected:

$$\bot \sqsubseteq x : \bot \sqsubseteq x : x : \bot \sqsubseteq x : x : x : \bot \sqsubseteq \ldots$$

The basic method for proving properties of programs defined in terms of the *fix* operator is Scott and de Bakker's *fixpoint induction* [8]. Suppose that f is a continuous function on a cpo and P is a predicate on the same cpo, such that whenever P holds of all the elements in a chain then it also holds of the limit (that is, P is a *chain complete* predicate.) Then the fixpoint induction rule can be stated as the following inference rule:

$$\frac{P\ \bot \qquad \forall x.\ P\ x \Rightarrow P\ (f\ x)}{P\ (\mathit{fix}\ f)}$$

This rule states that if the predicate holds of the least element $\bot$ of the cpo, and whenever it holds of an element x then it also holds of $f\ x$, then the predicate also holds of the least fixpoint *fix* f. The proof of the fixpoint induction rule is a simple consequence of the definition of *fix* as the limit of a chain of approximations. It is interesting to note that fixpoint induction is also the "free theorem" [33] for the *fix* operator.

As an example, let us see how fixpoint induction can be used to prove the property $map\ f \cdot iterate\ f = iterate\ f \cdot f$ from the previous section. First of all, we abstract from the occurrence of $iterate\ f$ and define a predicate $P\ g \Leftrightarrow map\ f \cdot g = g \cdot f$, so that the property to be proved can now be written as $P\ (iterate\ f)$. The predicate P is chain complete, because any predicate expressed as an equation between continuous functions is chain complete. Next, the meaning of *iterate* is given by $iterate\ f = fix\ (h\ f)$, where the auxiliary function h is defined as follows:

$$\begin{aligned} &h && :: (\alpha \to \alpha) \to (\alpha \to [\alpha]) \to \alpha \to [\alpha] \\ &h\ f\ g\ x && = x : g\ (f\ x) \end{aligned}$$

Hence, the property to be proved can be written as $P\ (fix\ (h\ f))$, which now matches the conclusion of the fixpoint induction rule, from which we conclude that the property is implied by the following two assumptions:

$$P\ (\lambda x.\bot) \qquad \forall g.\ P\ g \Rightarrow P\ (h\ f\ g)$$

Finally, these assumptions are verified by two calculations:

$$\begin{aligned} & P\ (\lambda x.\bot) \\ \Leftrightarrow\ & \quad \{\ \text{definition of } P\ \} \\ & map\ f \cdot \lambda x.\bot = \lambda x.\bot \cdot f \\ \Leftrightarrow\ & \quad \{\ \text{extensionality, composition}\ \} \\ & \forall y.\ map\ f\ ((\lambda x.\bot)\ y) = (\lambda x.\bot)\ (f\ y) \\ \Leftrightarrow\ & \quad \{\ \beta\text{-reduction}\ \} \\ & \forall y.\ map\ f\ \bot = \bot \\ \Leftrightarrow\ & \quad \{\ map\ f \text{ is strict}\ \} \\ & true \end{aligned}$$

and

$$\begin{aligned} & P\ (h\ f\ g) \\ \Leftrightarrow\ & \quad \{\ \text{definition of } P\ \} \\ & map\ f \cdot h\ f\ g = h\ f\ g \cdot f \\ \Leftrightarrow\ & \quad \{\ \text{extensionality, composition}\ \} \\ & \forall x.\ map\ f\ (h\ f\ g\ x) = h\ f\ g\ (f\ x) \\ \Leftrightarrow\ & \quad \{\ \text{definition of } h\ \} \\ & \forall x.\ map\ f\ (x : g\ (f\ x)) = f\ x : g\ (f\ (f\ x)) \\ \Leftrightarrow\ & \quad \{\ \text{definition of } map\ \} \\ & \forall x.\ f\ x : map\ f\ (g\ (f\ x)) = f\ x : g\ (f\ (f\ x)) \\ \Leftarrow\ & \quad \{\ \text{extensionality, composition, Leibniz}\ \} \\ & map\ f \cdot g = g \cdot f \\ \Leftrightarrow\ & \quad \{\ \text{definition of } P\ \} \\ & P\ g \end{aligned}$$

This completes the proof. Similarly, all the other properties from the previous section can also be proved using fixpoint induction. However, note that because fixpoint induction is an implication rather than an equivalence, it is sound but not complete, and is hence not always applicable. For example, with P defined as "is an infinite list" and $f\ x = 1 : x$, the statement $P\ (\mathit{fix}\ f)$ expresses that $1 : 1 : 1 : 1 : \ldots$ is an infinite list, but cannot be proved using fixpoint induction because the base case $P\ \bot$ is false.

4 The approximation lemma

Fixpoint induction is a rather basic proof method. In particular, it is tedious to have to return to first principles and perform proofs at the level of the fixpoint semantics of programs. Fortunately, for the special case of corecursive programs that produce lists, Bird and Wadler's *take lemma* [4] allows us to perform proofs at the level of the syntax of programs, without reference to their underlying fixpoint semantics.

Recently, the take lemma has been superseded by the *approximation lemma* [3], which is formally equivalent to the take lemma, but is easier to prove, slightly simpler to apply, and most importantly of all, naturally generalises from lists to a variety of other recursive types [12]. The basis of the approximation lemma is a function *approx*, defined as follows:

$$\begin{array}{ll} \mathit{approx} & :: \mathit{Int} \to [\alpha] \to [\alpha] \\ \mathit{approx}\ (n+1)\ [\,] & = [\,] \\ \mathit{approx}\ (n+1)\ (x : xs) & = x : \mathit{approx}\ n\ xs \end{array}$$

The *approx* function is similar to the well-known function *take*, except that there is no base case $\mathit{approx}\ 0\ [\,] = [\,]$, which by case exhaustion means that $\mathit{approx}\ 0\ xs = \bot$ for all lists xs. For example, $\mathit{approx}\ 4\ (\mathit{from}\ 0) = 0 : 1 : 2 : 3 : \bot$, while $\mathit{approx}\ 4\ (0 : 1 : 2 : [\,]) = 0 : 1 : 2 : [\,]$.

Suppose that xs and ys are two infinite, partial, or finite lists. Then the approximation lemma can be stated as the following equivalence:

$$xs = ys \quad \Leftrightarrow \quad \forall n.\ \mathit{approx}\ n\ xs = \mathit{approx}\ n\ ys$$

This equivalence states that two lists are equal precisely when all their approximations are equal. The left-to-right direction is trivially true by extensionality. For the other direction, it is easy to show that

$$\mathit{approx}\ 0 \sqsubseteq \mathit{approx}\ 1 \sqsubseteq \mathit{approx}\ 2 \sqsubseteq \mathit{approx}\ 3 \sqsubseteq \ldots$$

is a chain that has the identity function id as its limit (by induction on natural numbers and lists, respectively), using which result the right-to-left direction of the approximation lemma is proved as follows:

$$
\begin{array}{ll}
 & xs \;=\; ys \\
\Leftrightarrow & \quad \{ \text{ definition of } id \; \} \\
 & id\ xs \;=\; id\ ys \\
\Leftrightarrow & \quad \{ \text{ above result } \} \\
 & (\bigsqcup_n \{approx\ n\})\ xs \;=\; (\bigsqcup_n \{approx\ n\})\ ys \\
\Leftrightarrow & \quad \{ \text{ continuity of application } \} \\
 & \bigsqcup_n \{approx\ n\ xs\} \;=\; \bigsqcup_n \{approx\ n\ ys\} \\
\Leftarrow & \quad \{ \text{ property of limits } \} \\
 & \forall n.\ approx\ n\ xs \;=\; approx\ n\ ys
\end{array}
$$

As an example, let us see how $map\ f \cdot iterate\ f \;=\; iterate\ f \cdot f$ can be proved more simply by using the approximation lemma than by fixpoint induction. First of all, we use extensionality and the definition of composition to rewrite the property to be proved in the equivalent form:

$$\forall x.\ map\ f\ (iterate\ f\ x) \;=\; iterate\ f\ (f\ x)$$

Now, by the approximation lemma, this is equivalent to:

$$\forall x, n.\ \ approx\ n\ (map\ f\ (iterate\ f\ x)) \;=\; approx\ n\ (iterate\ f\ (f\ x))$$

Finally, this property is verified by induction on the natural number n. The base case $n = 0$ is trivially true because $approx\ 0\ xs = \bot$ for all xs. For the inductive case $n = m + 1$, we calculate as follows:

$$
\begin{array}{ll}
 & approx\ (m+1)\ (map\ f\ (iterate\ f\ x)) \\
= & \quad \{ \text{ definition of } iterate \; \} \\
 & approx\ (m+1)\ (map\ f\ (x : iterate\ f\ (f\ x))) \\
= & \quad \{ \text{ definition of } map \; \} \\
 & approx\ (m+1)\ (f\ x : map\ f\ (iterate\ f\ (f\ x))) \\
= & \quad \{ \text{ definition of } approx \; \} \\
 & f\ x :\ approx\ m\ (map\ f\ (iterate\ f\ (f\ x))) \\
= & \quad \{ \text{ induction hypothesis } \} \\
 & f\ x : approx\ m\ (iterate\ f\ (f\ (f\ x))) \\
= & \quad \{ \text{ definition of } approx \; \} \\
 & approx\ (m+1)\ (f\ x : iterate\ f\ (f\ (f\ x))) \\
= & \quad \{ \text{ definition of } iterate \; \} \\
 & approx\ (m+1)\ (iterate\ f\ (f\ x))
\end{array}
$$

This completes the proof. As well as leading to simpler proofs, because the approximation lemma is an equivalence it is both sound and complete, in contrast to the fixpoint induction rule which is only sound.

5 Bisimilarity and coinduction

Both fixpoint induction and the approximation lemma are based on the denotational approach to semantics. Another popular approach to semantics is the operational approach [26], for which a widely-used notion of equivalence for

programs is *bisimilarity*, and the basic proof method for establishing that two programs are bisimilar is *coinduction* [24]. In this section we review Gordon's approach [14] to applying bisimilarity and corecursion to prove properties of corecursive functional programs.

The operational semantics of a Haskell-like language can be defined by a reduction relation $\rightsquigarrow$, for which $a \rightsquigarrow a'$ means that the expression a can be reduced to the expression a' in a single execution step. For example, the following reductions formalise how *iterate* and *map* are executed:

$$\begin{array}{ll} \mathit{iterate}\ f\ x & \rightsquigarrow x : \mathit{iterate}\ f\ (f\ x) \\ \mathit{map}\ f\ [\,] & \rightsquigarrow [\,] \\ \mathit{map}\ f\ (x : xs) & \rightsquigarrow f\ x : \mathit{map}\ f\ xs \end{array}$$

Using $\rightsquigarrow$ we can define a labelled transition relation $\rightarrow$, for which $a \xrightarrow{o} a'$ means that the expression a immediately permits the *observation* o, which results in a making the transition to the expression a'. For example, the following transitions formalise the observations that can be made of a non-empty list, namely that we can observe its *head* and *tail*:

$$x : xs \xrightarrow{head} x$$

$$x : xs \xrightarrow{tail} xs$$

By repeated application of the transition relation $\rightarrow$, we can generate a (possibly infinite) transition tree that captures all possible sequences of observations for an expression. Informally, two expressions are *bisimilar* if their transition trees are identical when we ignore the programs at the nodes in the trees, and only consider the observations that label the edges. That is, two programs are bisimilar if they cannot be distinguished by an observer who has no knowledge of the internal details of the programs.

Formally, a *bisimulation* in this context is a relation R on expressions such that if $a\ R\ b$ then the following two conditions are satisfied:

Whenever $a \xrightarrow{o} a'$ there is some b' for which $b \xrightarrow{o} b'$ and $a'\ R\ b'$

Whenever $b \xrightarrow{o} b'$ there is some a' for which $a \xrightarrow{o} a'$ and $a'\ R\ b'$

That is, expressions that are related by a bisimulation permit the same observations, which result in transitions to related expressions. The above definition does not determine a unique relation, but it can be shown that there is always a greatest bisimulation under the inclusion ordering $\subseteq$ on relations, which relates as many expressions as possible subject to the two conditions above. The greatest bisimulation is denoted by $\sim$, and two expressions for which $a \sim b$ are called *bisimilar*. The fact that, by definition, all other bisimulations are included in the greatest bisimulation $\sim$ is known as *coinduction*, and forms a simple but powerful proof method.

In order to use coinduction to prove that two expressions a and b are bisimilar $a \sim b$, we must construct a bisimulation R for which $a\ R\ b$. Then, by coinduction we know that R is included in $\sim$, and hence because $a\ R\ b$ we conclude that $a \sim b$,

as required. That is, by coinduction the problem of showing that two expressions are bisimilar can be reduced to the problem of constructing a bisimulation that relates the two expressions.

As an example, let us prove that $map\ f \cdot iterate\ f \sim iterate\ f \cdot f$ using coinduction. First of all, we rewrite the property in the equivalent form:

$$\forall f, x.\ \ map\ f\ (iterate\ f\ x) \sim iterate\ f\ (f\ x)$$

Next, we construct a relation R that encodes this property:

$$R = \{(map\ f\ (iterate\ f\ x), iterate\ f\ (f\ x)) \mid f :: \alpha \to \alpha, x :: \alpha\}$$

It is easy to see that the relation R is not itself a bisimulation, but we will now show that the larger relation $R \cup \sim$ is a bisimulation. By a simple calculation, we need only consider the case of $a\ (R \cup \sim)\ b$, where $a = map\ f\ (iterate\ f\ x)$ and $b = iterate\ f\ (f\ x)$. From the definitions of *map* and *iterate* under the reduction relation $\leadsto$, we have that

$$a \leadsto^*\ f\ x : map\ f\ (iterate\ f\ (f\ x))$$
$$b \leadsto^*\ f\ x : iterate\ f\ (f\ (f\ x))$$

where $\leadsto^*$ is the reflexive and transitive closure of $\leadsto$. Now, using the fact that if $c \leadsto^* c'$ then $c \xrightarrow{o} c''$ is equivalent to $c' \xrightarrow{o} c''$, together with the definition of the transition relation $\to$ for the case of non-empty lists, we can enumerate all the possible transitions for a and b:

$$a \xrightarrow{head} f\ x \qquad (1)$$
$$a \xrightarrow{tail} map\ f\ (iterate\ f\ (f\ x)) \qquad (2)$$
$$b \xrightarrow{head} f\ x \qquad (3)$$
$$b \xrightarrow{tail} iterate\ f\ (f\ (f\ x)) \qquad (4)$$

Transition (1) is matched by transition (3) and vice versa, because $f\ x \sim f\ x$ by the reflexivity of $\sim$, which implies that $f\ x\ (R \cup \sim)\ f\ x$. Similarly, transition (2) is matched by transition (4) and vice versa, because $a'\ R\ b'$ by the definition of R, where $a' = map\ f\ (iterate\ f\ (f\ x))$ and $b' = iterate\ f\ (f\ (f\ x))$, which implies that $a'\ (R \cup \sim)\ b'$. This completes the proof that $R \cup \sim$ is a bisimulation, from which we conclude by coinduction that $R \cup \sim$ is included in $\sim$, and hence that R itself is included in $\sim$, which establishes the original property to be proved.

We conclude this section with two remarks about coinduction. First of all, because coinduction (as with induction) is an implication, it is sound but not complete. And secondly, while coinduction is an elegant proof principle, denotational semantics is still the dominant basis for proofs about functional programs, and it seems unfortunate to have to change to the lower level of an operational semantics in order to exploit coinduction.

6 Unfold and universality

Others [23] have observed that arbitrary recursion is the goto of functional programming. A well-structured functional program is one in which the recursion is encapsulated in a small collection of higher-order operators, which are then used and reused throughout the text of the program (or the career of the programmer). The benefits of structure in functional programming are the same as the benefits of structure in imperative programming: programs are easier to write, to read and to reason about and less prone to errors, and a decent compiler can regain any loss of performance that naive implementation of the structured features might incur.

For example, the disciplined functional programmer will eschew the explicit recursion used in the definitions of *iterate* and *map* above, and will prefer instead to define a higher-order operator to encapsulate those recursion patterns and use that operator to define the two functions. The natural encapsulation of coinductively recursive functions is the 'unfold' [6, 13, 16] (or 'anamorphism' [23]). Unfold on lists is given by

$$\begin{aligned} &\mathit{unfold} && :: (\alpha \to \beta) \to (\alpha \to \alpha) \to \alpha \to [\beta] \\ &\mathit{unfold}\ f\ g\ x = f\ x : \mathit{unfold}\ f\ g\ (g\ x) \end{aligned}$$

Informally, $\mathit{unfold}\ f\ g$ generates a list from the 'seed' x. The head of this list is $f\ x$, and the tail of this list is generated recursively in the same way from the new seed $g\ x$. Thus,

$$\begin{aligned} &\mathit{iterate}\ f = \mathit{unfold}\ \mathit{id}\ f \\ &\mathit{map}\ f \quad = \mathit{unfold}\ (f \cdot \mathit{head})\ \mathit{tail} \end{aligned}$$

The disciplined functional programmer might also expect to derive some benefit from the structure inherent in their programs when it comes to proving properties of these programs. Unfortunately, none of the proof rules above can capitalise directly on good program structure. (There may be some *indirect* benefit from good program structure, as this might simplify the proof obligations generated by application of the proof rules; however, there is no gain at the level of the proof rules themselves.)

Fortunately, unfold enjoys a *universal property*, stating that it is the *unique solution* to a particular equation. The universal property of unfold is

$$h = \mathit{unfold}\ f\ g \quad \Leftrightarrow \quad \mathit{head} \cdot h = f \ \wedge\ \mathit{tail} \cdot h = h \cdot g$$

The implication from left to right is easy to verify; thus, $\mathit{unfold}\ f\ g$ is indeed a solution of the two equations in h on the right. It is not hard to convince oneself that the two equations on the right have in fact a *unique* solution, namely the unfold; the proof is more involved, and depends on showing that regular type functors (those involving sums and products, constant and identity functors, and recursion) in the appropriate category (of continuous functions between cpos) have terminal coalgebras; see [27, 21, 30] for the details. This universal property turns out to be extremely convenient for proving properties of unfolds.

As an example, let us see how $map\ f \cdot iterate\ f\ =\ iterate\ f \cdot f$ can be proved using this universal property. We show first a *fusion theorem*, giving conditions under which an unfold fuses with another function to form a new unfold:

$$
\begin{array}{ll}
 & unfold\ f\ g \cdot h = unfold\ f'\ g' \\
\Leftrightarrow & \quad \{\ \text{universal property}\ \} \\
 & head \cdot unfold\ f\ g \cdot h = f' \wedge tail \cdot unfold\ f\ g \cdot h = unfold\ f\ g \cdot h \cdot g' \\
\Leftrightarrow & \quad \{\ unfold, head \text{ and } tail\ \} \\
 & f \cdot h = f' \wedge unfold\ f\ g \cdot g \cdot h = unfold\ f\ g \cdot h \cdot g' \\
\Leftarrow & \quad \{\ \text{Leibniz}\ \} \\
 & f \cdot h = f' \wedge g \cdot h = h \cdot g'
\end{array}
$$

From this fusion theorem we obtain the special case of *map-unfold fusion*:

$$
\begin{array}{ll}
 & map\ h \cdot unfold\ f\ g \\
= & \quad \{\ \text{definition of } map\ \} \\
 & unfold\ (h \cdot head)\ tail \cdot unfold\ f\ g \\
= & \quad \{\ \text{fusion: } tail \cdot unfold\ f\ g = unfold\ f\ g \cdot g\ \} \\
 & unfold\ (h \cdot head \cdot unfold\ f\ g)\ g \\
= & \quad \{\ head \cdot unfold\ f\ g = f\ \} \\
 & unfold\ (h \cdot f)\ g
\end{array}
$$

Now

$$
\begin{array}{ll}
 & iterate\ f \cdot f \\
= & \quad \{\ \text{definition of } iterate\ \} \\
 & unfold\ id\ f \cdot f \\
= & \quad \{\ \text{fusion}\ \} \\
 & unfold\ f\ f \\
= & \quad \{\ \text{map-unfold fusion}\ \} \\
 & map\ f \cdot unfold\ id\ f \\
= & \quad \{\ \text{definition of } iterate\ \} \\
 & map\ f \cdot iterate\ f
\end{array}
$$

The proof using the universal property of unfold is much simpler than the proofs using any of the other three methods. In particular, the proof is conducted via simple equational reasoning at the level of the programs themselves, without having to descend to the level of the semantics (as with fixpoint induction) or translate to a completely different formalism (as with bisimulation). Moreover, every regular type definition yields an unfold and a corresponding universal property for free [23]; no extra facts need to be proved for each new type (as they do with the take and approximation lemmas).

As a further example, consider the function

$$
\begin{array}{l}
tails :: [\alpha] \to [[\alpha]] \\
tails = unfold\ id\ tail
\end{array}
$$

One fact about *tails* is that

$$map\ tails \cdot tails = tails \cdot tails$$

The proof is very straightforward:

$$
\begin{array}{ll}
 & tails \cdot tails \\
= & \quad \{ \text{ definition of } tails \ \} \\
 & unfold\ id\ tail \cdot tails \\
= & \quad \{ \text{ fusion: } tail \cdot tails = tails \cdot tail \ \} \\
 & unfold\ tails\ tail \\
= & \quad \{ \text{ map-unfold fusion } \} \\
 & map\ tails \cdot unfold\ id\ tail \\
= & \quad \{ \text{ definition of } tails \ \} \\
 & map\ tails \cdot tails
\end{array}
$$

Gordon [14] proves a similar result (actually, a kind of dual of this result, that

$$concat \cdot map\ concat = concat \cdot concat$$

but the steps are the same) using bisimulation; his proof is significantly more complicated than this.

7 Summary and conclusions

We have discussed a number of proof methods for corecursive programs, and argued that equational reasoning using a universal property is the most convenient. Certainly, our proofs using the universal property are simpler than Gordon's proof using bisimulation [14], and more convincing than Dijkstra's [9, 10], of the same properties.

We have only discussed unfolds for lists, which is about the simplest recursive type possible. However, there is nothing special about lists; it is straightforward (and even automatable [19]) to define unfolds for arbitrary regular types; see [23] for a summary of the details. Everything we have said about unfolds for lists holds also for unfolds for other types.

Of course, not all corecursive functions are unfolds; the pattern is a little restrictive. But neither are unfolds the only useful encapsulation of corecursion. Uustalu and Vene [32] dualise Meertens' paramorphisms [22] to obtain what they call *apomorphisms*. Informally, an apomorphism is like an unfold, but with the possibility of 'escape clauses' in which the remainder of the result is constructed immediately, without further recursion. Apomorphisms are a generalisation of unfolds; in fact, it turns out that every function whose target type is a recursive type can be written as an apomorphism. Moreover, apomorphisms too enjoy a universal property, every bit as powerful as the universal property of unfolds.

Acknowledgements

The second author is supported by EPSRC grant *Structured Recursive Programming*, and ESPRIT Working Group *Applied Semantics*.

References

1. Peter Aczel. *Non-Well-Founded Sets*. Number 14 in CSLI Lecture Notes. Stanford: CSLI Publications, 1988.
2. Jon Barwise and Lawrence Moss. *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena*. Number 60 in CSLI Lecture Notes. Stanford: CSLI Publications, 1996.
3. Richard Bird. *Introduction to Functional Programming using Haskell (second edition)*. Prentice Hall, 1998.
4. Richard Bird and Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988.
5. W.H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.
6. Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, University of Calgary, 1992.
7. B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
8. Jaco de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall, 1980.
9. E. W. Dijkstra. A very simple exercise in SASL. EWD827, 1982.
10. E. W. Dijkstra. Proving the equality of infinite sequences. EWD1093, 1991.
11. Peter Freyd. Algebraically complete categories. In A. Carboni et al, editor, *Proc. 1990 Como Category Theory Conference*, volume 1488 of *Lecture Notes in Math*, pages 95–104. Springer-Verlag, Berlin, 1990.
12. Jeremy Gibbons and Graham Hutton. The generic approximation lemma. In preparation, 1999.
13. Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proc. 3rd ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Maryland, 1998.
14. Andrew Gordon. Bisimilarity as a theory of functional programming. BRICS Notes Series NS-95-3, Aarhus University, 1995.
15. Graham Hutton. A tutorial on the universality and expressiveness of fold. To appear in the Journal of Functional Programming.
16. Graham Hutton. Fold and unfold for program semantics. In *Proc. 3rd ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Maryland, 1998.
17. Bart Jacobs, Larry Moss, Horst Reichel, and Jan Rutten, editors. *Proc. of the First Workshop on Coalgebraic Methods in Computer Science*. Elsevier Science B.V., 1998. Electronic Notes in Theoretical Computer Science Volume 11.
18. Bart Jacobs and Jan Rutten, editors. *Proc. of the Second Workshop on Coalgebraic Methods in Computer Science*. Elsevier Science B.V., 1999. Electronic Notes in Theoretical Computer Science Volume 19.
19. Patrik Jansson and Johan Jeuring. PolyP - a polytypic programming language extension. In *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1997.
20. Simon Peyton Jones et al. Haskell 98: A non-strict, purely functional language. Available on the World-Wide-Web from `http://www.haskell.org`, February 1999.
21. Daniel J. Lehmann and Michael B. Smyth. Algebraic specification of data types: A synthetic approach. *Mathematical Systems Theory*, 14:97–139, 1981.
22. Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.

23. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Proc. Conference on Functional Programming and Computer Architecture*, number 523 in LNCS. Springer-Verlag, 1991.
24. Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
25. Lawrence Moss and Norman Danner. On the foundations of corecursion. *Logic Journal of the IGPL*, 5(2):231–257, 1997.
26. Gordon Plotkin. A structured approach to operational semantics. Report DAIMI-FN-19, Computer Science Department, Aarhus University, Denmark, 1981.
27. John C. Reynolds. Semantics of the domain of flow diagrams. *Journal of the ACM*, 24(3):484–503, 1977.
28. John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.
29. David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.
30. Mike B. Smyth and Gordon D. Plotkin. The category theoretic solution of recursive domain equations. *SIAM Journal of Computing*, pages 761–783, 1982.
31. David A. Turner. Elementary strong functional programming. In *Proc. First International Symposium on Functional Programming Languages in Education*, LNCS 1022, pages 1–13. Springer-Verlag, 1995.
32. Varmo Vene and Tarmo Uustalu. Functional programming with apomorphisms (corecursion). In *9th Nordic Workshop on Programming Theory*, Oct 1997.
33. Philip Wadler. Theorems for free! In *Proc. Conference on Functional Programming and Computer Architecture*. Springer, 1989.